ESC-TR-2001-086

Project Report
ECCS-1

Declarative Routing Protocol Documentation

P. Boettcher
D. Coffin
R. Czerwinski
K. Kurian
M. Nischan
D. Sachs
G. Shaw
D. Van Hook

28 February 2003

Lincoln Laboratory
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
LEXINGTON, MASSACHUSETTS

Prepared for the Defense Advanced Research Projects Agency
under Air Force Contract F19628-00-C-0002.

This report is based on studies performed at Lincoln Laboratory, a center for research operated by Massachusetts Institute of Technology. The work was sponsored by the Defense Advanced Research Projects Agency, IXO, under Air Force Contract F19628-00-C-0002.

Massachusetts Institute of Technology
Lincoln Laboratory

Declarative Routing Protocol Documentation

P. Boettcher
K. Kurian
G. Shaw
Group 99

R. Czerwinski
M. Nischan
Group 97

D. Van Hook
Group 65

D. Coffin
BBN

D. Sachs
Coordinated Science Laboratory
University of Illinois

Project Report ECCS-1

28 February 2003

Lexington Massachusetts

ABSTRACT

This report documents the motivation, present capability, and theoretical promise of the Declarative Routing Protocol (DRP) developed at MIT Lincoln Laboratory as part of the DARPA Sensor Information Technology (SensIT) program. DRP was developed as a means of enabling distributed wireless sensors to configure themselves into a scalable ad hoc network and respond in an energy-efficient way to asynchronous requests for sensor information. Conventional networking approaches are generally not adequate for such applications because of energy constraints, reliability and scalability requirements and the greater variability in topology present compared with traditional fully-wired or last-hop wireless (remote to base station) networks. DRP operates within these constraints by exploiting query-supplied data descriptions to control network routing and resource allocation.

ACKNOWLEDGMENTS

This work has been carried out by MIT Lincoln Laboratory under contract to the Air Force and sponsorship by DARPA/ITO. Dr. Sri Kumar, SensIT Program Manager, DARPA/ITO, has provided the guidance.

TABLE OF CONTENTS

Preceding Page Blank

LIST OF ILLUSTRATIONS

1. INTRODUCTION

1.1. Background

The Declarative Routing Protocol (DRP) was developed under the DARPA Sensor Information Technology (SensIT) program [1]. This program supports research in information technology and distributed sensing, including projects in the areas of network routing, collaborative sensing and database technology.

The purpose of this report is to describe a declarative approach to network configuration and organization that appears to offer significant benefits for deployment of collaborative sensors in military applications. In this application, a number of constraints interact to make sensor networking difficult. Some of these constraints are:

- Sensors must be low cost. They are viewed as consumable assets that are deployed, perform their function, and are left behind or replaced when their energy is depleted. Consequently, they must be relatively simple and will have limited capabilities.

- Sensors must be autonomously networked. No inherent network organization or in–place communication infrastructure can be assumed to exist. Communication between sensors and with users is via ad hoc wireless networks.

- The network topology is dynamic and generally unknown to the user, who formulates queries based on interest in specific geographical regions and data types.

- Sensor networks must be scalable. Large numbers (perhaps up to tens of thousands) of nodes may be deployed.

- Sensors must collaborate to reduce their up-stream bandwidth requirements. Throughput may be limited by power constraints, network reliability and by requirements for jam resistance, security, and covertness.

- Low latency may be necessary, especially for applications that include weapons targeting.

- Sensor nodes are energy constrained. They are compact and may need to function covertly behind enemy lines for extended time periods. As a consequence, energy efficiency is a major issue.

- Nodes may be mobile and may be deployed at different times. They must, however, still be integrated into a coherent, collaborating sensor system.

Our goal is to leverage concepts developed for wireless networking that fit within these constraints. Declarative routing is related to a number of such concepts. For example, the Destination-Sequenced Distance Vector (DSDV) algorithm [9] limits network latency by creating a priori routing tables. However, this approach relies on global interactions to set up routing, and does not scale to the number of sensor nodes envisioned. In any event DSDV requires routing tables to be updated continually as topology changes.

Ad hoc On Demand Distance Vector (AODV) routing [10], the Dynamic Source Routing (DSR) protocol [11] and the Temporally-Ordered Routing Algorithm (TORA) [12] all provide on-demand routing more suitable to a mobile environment. Our declarative approach shares some similarities with these approaches. However declarative routing represents a distinct paradigm in communication because of two significant differences. First, in declarative routing, the data consumer (not producer) initiates the routing, by declaring interest in the data. Second, because routes are established based on expressed interest, routing can be directed away from nodes with no potential to produce corresponding data (i.e. those outside the specified geographical area.)

Another approach to routing is the GeoCast effort [13], in which routers are set up based on geographic locations. While this solution is not optimized for energy efficiency, it shares some features with our declarative routing approach (e.g. routing based on declarations of location). Finally, the Directed Diffusion [2, 3] approach under development shares many similarities with our declarative approach.

1.2. Declarative Networking Core Concepts

Sensor network communication is often dominated by many-to-one data transfers in which specific types of data from all available source nodes within some geographic region must be relayed, for example alerts of an intruder detection. Supporting such applications using point-to-point addressing and route-building approaches is wasteful of transmit energy and may become unreliable in an ad hoc and highly dynamic wireless environment. There are two core concepts that distinguish declarative routing and make it more suitable.

Subscription / Publication Networking Paradigm. Declarative networking is based on a different paradigm from IP-style address-based networking. In declarative networking, applications *declare* their interest in specific data (subscriptions) or the ability to produce data (publications). The subscriptions are then used to create routing trees to potential producers. Subscriptions and publications describe data in terms that can include the type of data, range, size, geographic location, reliability, rate, desired latency, fidelity, and resolution. In essence, subscriptions and publications define and describe the application data flow. Thus, data declarations constitute an excellent basis on which to organize and configure a network of collaborative sensors.

Figure 1-1 shows an example of a publication and subscription. In this example, the publication matches the subscriber's interest, so data is routed to the subscriber. Data declarations can be exploited at multiple layers to benefit routing, transmission power control, smart antenna steering, and transmission scheduling. Subscription and publication types will be discussed in Section 2 of this report.

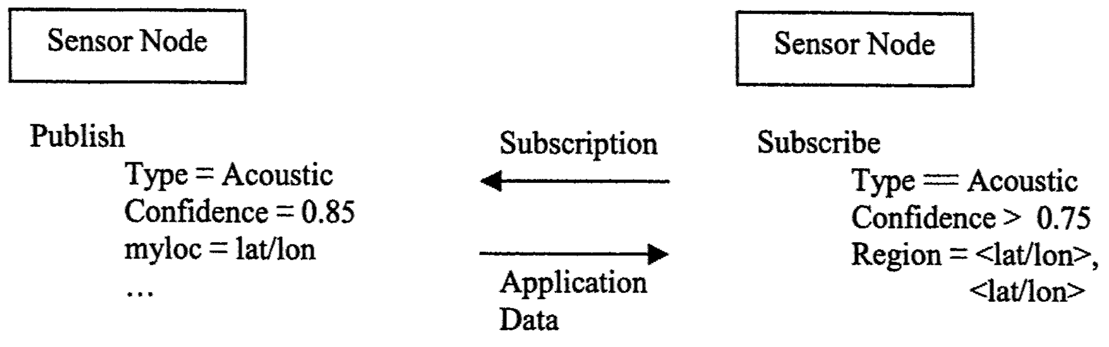

Figure 1-1: Example Publication and Subscription

Sharing Information across Layers. In the evolution of computer networking, layered network architectures were developed to isolate network functions and allow a high degree of modularity and interchangeability. However in an energy-constrained ad hoc network, the layers may make poor or conflicting resource tradeoffs, for example by establishing or maintaining routes to nodes with which the application layer has no need to communicate. Furthermore, bandwidth and energy-constrained collaborative sensing applications can not afford the duplicate computation and other inefficiencies inherent in the layered network approach.

A fundamental concept of the DRP implementation is that key information can be exploited by sharing information across the traditional layering boundaries. Figure 1-2 shows a new entity, *declaration services,* that spans the other more familiar layers, making key information available across layers to enable more nearly optimal system-wide decisions. A significant challenge is to develop such an approach while maintaining modularity, extensibility, and testability.

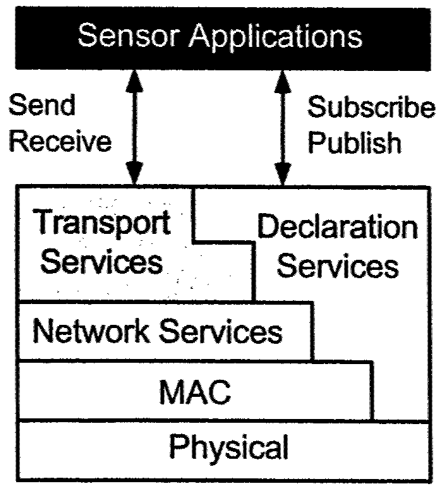

Figure 1-2: Declarative Networking

1.3. Declarative Routing Protocol Benefits

Beyond the core concepts presented in the previous section, DRP is distinguished by a number of other important characteristics, as described in the following paragraphs.

Geography-based routing. DRP exploits location information as it establishes routes. Applications may subscribe for data from geographic regions they specify. DRP uses geographic declarations to selectively form routes that are geographically rather than IP driven.

*Local computations on local dat*a. DRP exploits only local information at each node, resulting in substantial scalability and low energy consumption.

Distributed, replicated computations. DRP employs distributed, replicated computations in which nodes independently calculate equivalent routing solutions based on consistent views of a distributed database of node state. This approach trades off increased computation for reduced communication. This is important because the energy cost of computation is dropping much more rapidly than the energy cost of communication [4].

*Power managemen*t. The DRP concept is compatible with node power management approaches such as powering down relatively energy-inefficient RF hardware and signal processors during periods when no activity is sensed. This can be important, since significant energy can be expended by idle nodes, especially those with RF hardware [8].

*Energy aware routin*g. DRP routing calculations can be made to include consideration of available node energy (an idea that is also presented in [8]). Nodes with substantial energy reserves can be favored for routing over those that are lacking in energy. This may result in routing changes similar to those induced by physical mobility or changes in link quality, and can extend system life by reducing the overuse of critical nodes.

*Redundant routin*g. DRP supports the establishment and maintenance of multiple routes for purposes of improved reliability. Multiple routes can be used to improve robustness through multiple sends, striping, and random routing. In addition, nodes serve as backup routers that forward data after an interval if they do not detect a transmission from a primary routing node.

Mobility. DRP supports mobile nodes by allowing them to remain connected through whichever nodes are presently nearby. This support comes at the cost of increased energy consumption, however the cost is incurred only when a node moves, and it is localized to surrounding nodes. Except in cases of very frequent node mobility, it is not expected that this overhead will dominate the total energy consumption.

2. DECLARATIVE ROUTING PROTOCOL DESCRIPTION

The Declarative Routing Protocol (DRP) concept is based on the formation of *neighborhoods* of nodes in a distributed fashion; these neighborhoods direct information from the source of information to the consumer. A neighborhood is defined as a set of nodes that can communicate bi-directionally via a single hop with some minimum acceptable probability of success. Neighborhood membership does not imply that a node will successfully receive *every* message sent by every neighboring node; it does imply, however, that neighbors can engage in useful and effective communication. A key feature in the DRP neighborhood concept is that there is no single neighborhood head and hence no protocols to transfer leadership if a head node is destroyed or moves.

As an illustration of this concept, consider the collection of wireless nodes shown in Figure 2-1.

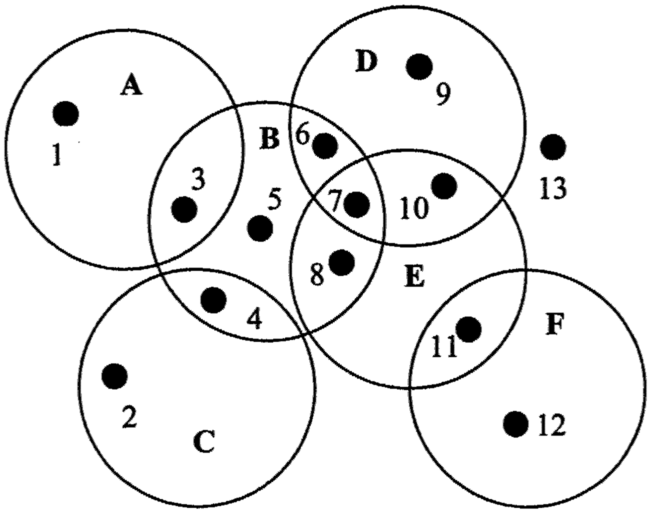

Figure 2-1: A Collection of Wireless Nodes

In this figure, nodes distributed over some geographic area are depicted as numbered dots. Circles indicate neighborhoods of nodes that can communicate with each other. Nodes within each neighborhood can send to and receive from all other nodes in the same neighborhood. For example, node 11 can exchange data with node 10. Similarly, node 10 can exchange data with node 11. The same bi–directional relationship does not exist between nodes 11 and 9, however, so they are not shown in the same circle.

In the physical world, neighborhoods of nodes capable of bi–directional communication are not generally circular. The actual shape of such neighborhoods is a function of many factors including topography, location of interference sources, antenna characteristics, and variations in individual node transceiver performance. In DRP, the neighborhoods are formed implicitly, i.e., nodes identify their neighbors by successfully exchanging their node states.

2.1. Distributed Node State

The nodes in a wireless network together form a distributed database of node state data that supports network-level operations including neighborhood formation and maintenance, route calculation, and data forwarding. The node state record for each node includes the following components:

- Node ID and location
- List of neighboring nodes
- Information about each subscription including:
  - Subscription handle and owner (subscriber node ID)
  - Data specification provided by the subscribing node, including geographic regions, data type of interest (e.g. raw acoustic data), and any other element the subscriber would like to filter on.
  - List of neighboring nodes from which this node will receive routed data (for this subscription).
  - Single neighboring node to which this node will forward routed data (for this subscription).

The node state evolves over time, and each time that it changes, a node state sequence number is incremented. The sequence number is also kept as part of node state and is exchanged with neighbors to help manage their remote records of the node state.

Each individual node also caches part of the node state records originating from neighbors. Data cached from neighbors is limited to:

- Neighbor node ID
- Neighbor node location
- Neighbor node state sequence number
- List of neighbor subscriptions
- List of neighbor's neighbors.

Figure 2-2 illustrates the concept of the distributed node state database. This figure depicts five wireless nodes, numbered 1 through 5. Bi–directional connectivity exists on a pair-wise basis between nodes 1, 2, and 3 as indicated by double headed arrows. Consequently, each of these nodes caches the node state of the others. Node 4, however, has bi–directional connectivity only with node 3. Additionally, node 5 has only unidirectional connectivity with node 4. The single-headed arrow connecting nodes 4 and 5 indicates that node 5 receives data from node 4 but that the converse is not true[1].

These relationships cause each node to cache different subsets of the full distributed node state database. The node state versions cached for each neighbor node are indicated by the numbered boxes. The complete distributed node state database does not exist at any single node. Instead, it is distributed across the system as the union of the caches of all

[1]Note that node 5 could purge node 4 from its list of neighbors since it never sees itself among node 4's neighbors. In practice, one-directional connectivity is rare and DRP does not implement such a check.

the nodes. As the connectivity changes, i.e., as nodes come into and go out of communication with each other, the subset of node state records cached by the affected nodes is altered.

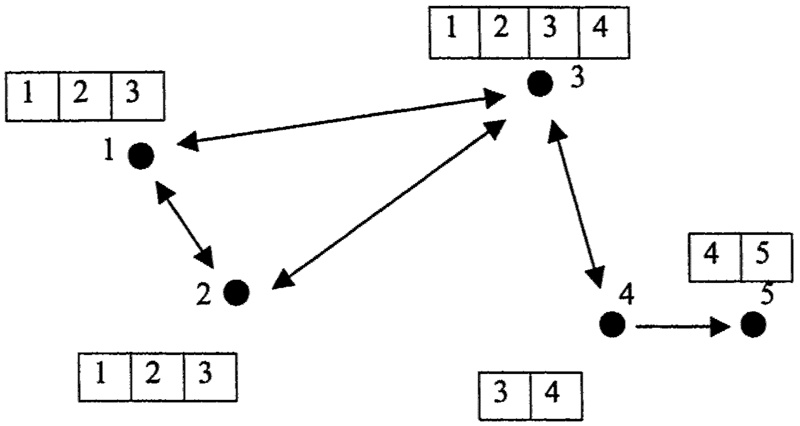

Figure 2-2: Distributed Node State Database

When a node changes state, it increments its node state sequence number and sends its updated node state record to all neighbors. Updates to neighbors' node state are required only under very specific circumstances – a new neighbor or a new subscription (or unsubscribe). Because of the uncertain reliability of wireless media, however, the node state update may not always be received.

To ensure all nodes maintain a consistent view of the distributed node state database, a *consistency protocol* is implemented. Each node sends small beacon messages at a predetermined interval once the node state record has become quiescent. This beacon contains only the identity of the originating node and the sequence number of its latest node state record. The beacon can be piggybacked on top of other transmissions when possible.

Remote nodes, upon receiving the beacon signal, compare the transmitted node state sequence number with their own cached record. If a node detects an inconsistency, it requests the latest version of the state by issuing a *NAK* message to the originating node. The originating node responds by re-transmitting the entire node state record. If bandwidth is limited, network traffic can be lessened by updating only the changed elements of node state – rather than the entire record – at the cost of increased overhead storage by each node of all previous state updates.

The following figures depict the consistency protocol's operation. These figures show three wireless nodes that have received and maintained node state information from each other. One node (node 3) has missed a transmission and has an older version of the node state from node 2. Assuming the nodes can communicate reliably (i.e. they are truly neighbors), node 3 ultimately obtains a current copy of the state of node 2 as shown in the figure.

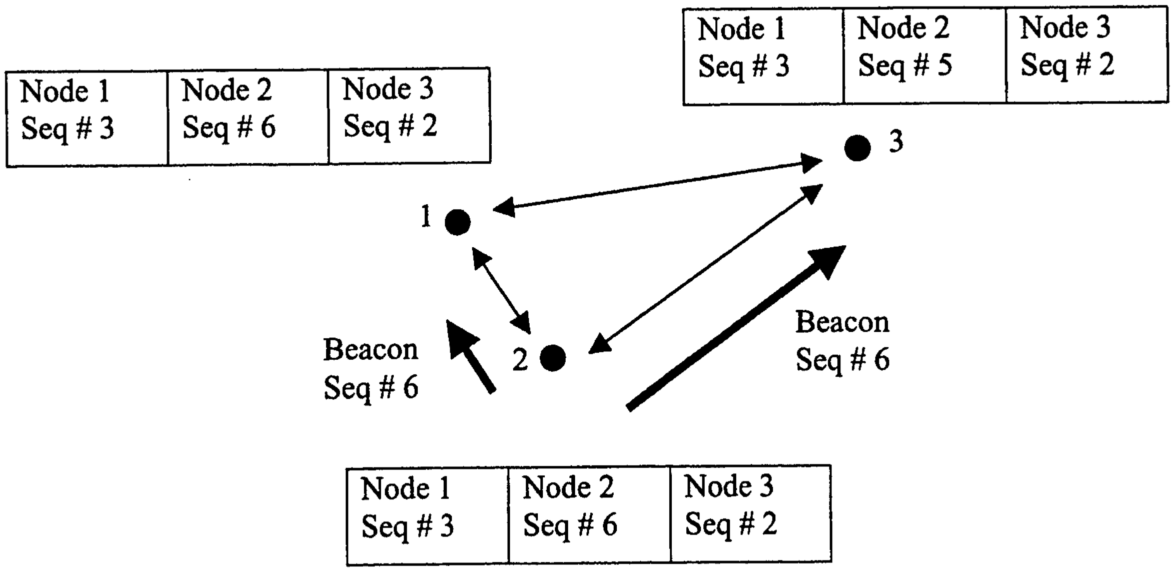

Figure 2-3a: Node 2 sends a beacon containing sequence number 6

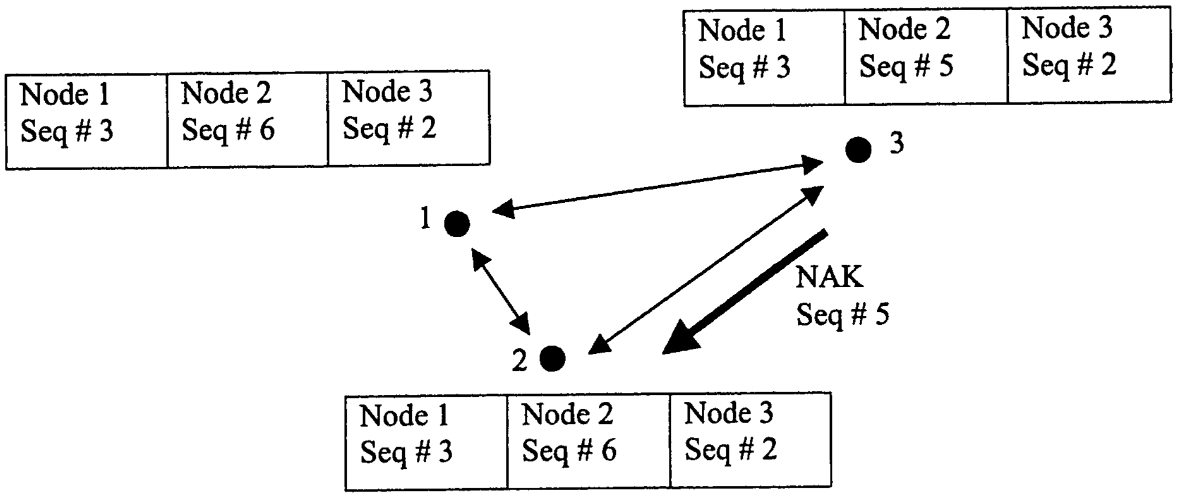

Figure 2-3b: Node 3 does not have correct version of node state for node 2 and sends a NAK.

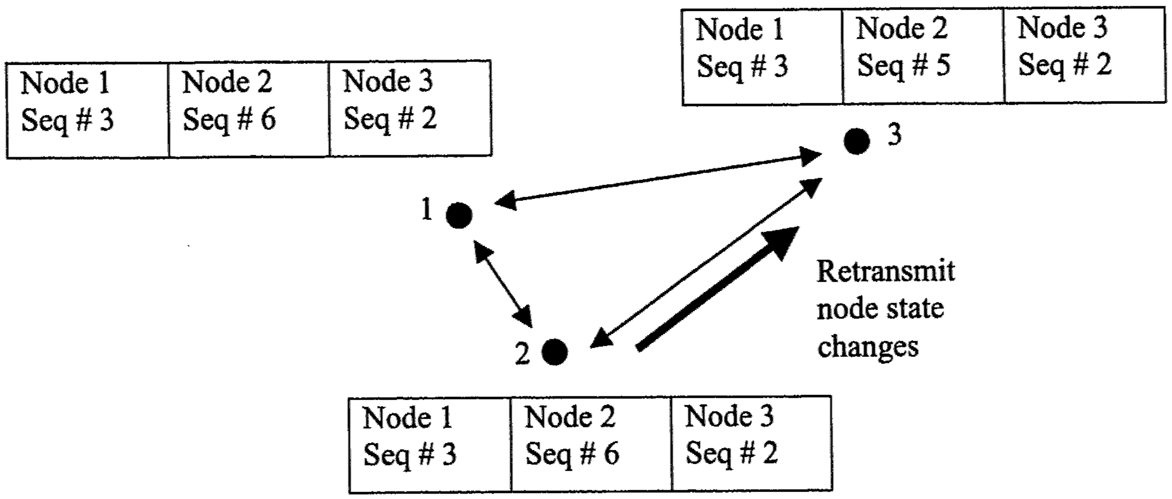

Figure 2-3c: Node 2 retransmits state changes associated with sequence number 5 and 6.

This consistency protocol is also involved in eliminating nodes from a neighborhood when they move or are destroyed (and fail to send a heartbeat message for some length of time), and in initializing the network when many nodes initially establish connectivity. In this case, to avoid overloading the network with multiple simultaneous requests for state updates, nodes can snoop the responses to their neighbors' update requests.

2.2. Subscription Propagation

Within a neighborhood, producers can send data directly to consumers. For example, node 8 in Figure 2-1 can communicate directly with nodes 3, 4, 5, 6, 7, 10, and 11. This is true because node 8 shares membership in neighborhood B or E with these other nodes. However, node 3 cannot send data directly to node 11. Instead, data must be forwarded along a route consisting of neighborhood–to–neighborhood hops with intermediate nodes serving as forwarders.

Routing of a subscription is based on a subscriber's request (i.e. subscription) for data of a particular type from a particular region. A subscription is uniquely identified by two components of the node state database: the subscriber node ID and the subscription handle. This information prevents loops in the routes and allows multiple routes to exist between the publisher and subscriber.

When the subscription is declared, potential routes are established by a distributed algorithm that forms a tree directed towards (and ultimately contained within) the desired geographic region. Because the subscription is directed, the network is prevented from propagating subscription information to nodes in undesired geographical directions. Once a subscription reaches a node, it persists indefinitely or until the subscriber cancels the subscription. Route information is stored as part of the node state database and is maintained by the consistency protocol.

When a subscription has entered a neighborhood by becoming part of the node state of at least one member node, any publishing node in the neighborhood with matching data can respond by broadcasting packets tagged for the appropriate subscription. The packets are then relayed to the subscriber along the tree that was formed when the subscription was declared.

As an example of publication / subscription-based routing, consider the topology of wireless nodes with neighborhoods as shown in Figure 2-4. Assume that node 1 has subscribed for some particular type of data causing the formation of a routing tree originating at node 1. Boundary nodes with membership in multiple neighborhoods extend the tree between neighborhoods. In this example, the tree grows through adding neighborhood B to the tree after one hop, then adding neighborhoods C, D, and E after two hops and lastly adding neighborhood F after three hops. As long as the routing tree has entered the neighborhood, it is available for any publishers that are located in that neighborhood (e.g. the network depicted can access publications from node 12 in neighborhood F even though the subscription has propagated only as far as the edge of neighborhood F). The resulting routing tree is shown in Figure 2-4.

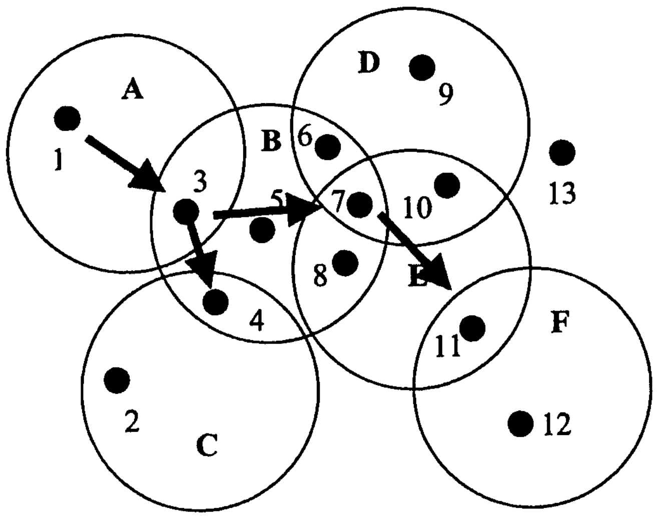

Figure 2-4: Generation of Routing Tree

When a subscription enters a neighborhood, it does so by first becoming a part of the node state record of the neighbor node selected to forward into that neighborhood. Then, because that node's state has changed, it sends an update to all its neighbors to make them aware of the subscription. Because all neighbors have a consistent record of the states of the nodes in the neighborhood, each is able to calculate in a consistent way which nodes are along the edges of new neighborhoods and which of these should further propagate the subscription.

This calculation has two parts. First, all the neighbors independently compile a subset of neighbors which is able to communicate with all two-hop neighbors. Second, a filter is applied to prevent nodes from forwarding the subscription unless they are aligned toward the region of interest. The initial subset is constructed by including in descending order of their score computed as a weighted average of the following components:

• **Reachability** – Reachability is a number between 0 and 1 computed as the number of nodes 2 hops away that a node can reach divided by the number of unique nodes 2 hops away that the entire neighborhood can reach. Note that all nodes in a neighborhood do not necessarily have a combined reachability of 1.0 (because of overlapping neighborhoods). For example, in the network of Figure 2-4, if a subscription is propagated from node 1, it propagates to node 3 in neighborhood B. Neighborhood B considers propagating the subscription through four edge nodes, numbered 4, 6, 7, and 8. Node 4 can reach node 2. Node 6 can reach nodes 9 and 10. Node 7 can reach nodes 9 and 10, and node 8 can reach nodes 10 and 11. The reachability score for node 4 is 1/5, node 6 is 2/5, node 7 is 2/5, and node 8 is 2/5.

• **Diversity** – Diversity is a number between 0 and 1 that measures the uniqueness of the nodes reachable by a particular edge node. The diversity score is defined for each boundary node as

$$D_i = \frac{1}{N}\sum_{j=1}^{N} d_{ij},$$

the average of the diversity measures computed pair-wise with every other candidate boundary node according to:

$$d_{ij} = \frac{(N_i + N_j - N_{ij}) - (N_{ij})}{(N_i + N_j - N_{ij})},$$

where N_i is the number of nodes reachable by node i, N_j is the number of nodes reachable by node j and N_{ij} is the number of nodes reachable by both node i and node j. For example, in the example above, the diversity scores for nodes 4, 6, 7, and 8 are given by:

node 4: [((3-0)/3) + ((3-0)/3) + ((3-0)/3)] / 3 = 1.00
node 6: [((3-0)/3) + ((3-2)/3) + ((3-1)/3)] / 3 = 0.66
node 7: [((3-0)/3) + ((3-2)/3) + ((3-2)/3)] / 3 = 0.55
node 8: [((3-0)/3) + ((3-1)/3) + ((3-2)/3)] / 3 = 0.66.

• **Directionality** – Directionality is number between –1 and 1 that measures a node's alignment along the shortest path to the geographic region of interest. This is implemented as

$$\frac{v_1 \bullet v_2}{v_1 \times v_2}$$

where v_1 denotes the vector from the subscribing node to the region of interest, v_2 denotes the vector from the subscribing node to the boundary node under consideration, and $\bullet$ and $\times$ denote vector dot- and cross-products respectively. This measure is the tangent of the angle difference between v_1 and v_2. This term is normalized to lie between –1 and 1 by simply clipping its value at those extremes. A negative value indicates a direction away from the area of interest. Additional capability may be achieved in future implementations by considering such factors as:

- **Energy Level** – Measure of how often a node is used based on battery power level
- **Power Dissipation** – Measure of future energy level based on present utilization
- **Connectivity** – Predicted stability based on the length of time a node has been connected
- **Link quality** – Signal to noise ratio or packet error rate

Note that because of the directionality term, the nodes chosen are not guaranteed to be the smallest possible subset that achieves connectivity with all two-hop nodes.

Once a subset of nodes is identified that is able to reach all two-hop neighbors, a second filter is applied to determine forwarding nodes based on geographical location, either within the region of interest or within the cone beginning at the forwarding node and including the edges of the region of interest plus some additional buffer angle[2]. The logic implemented in this filter is depicted in Figure 2-5, in which node 3 relays a subscription for data originating in the box. Only nodes within the box or cone may forward the subscription. In this case, only nodes 7 and 8 are within the cone and reachable by node 3, so they alone may be considered. Since they have the same neighbors within the box, only one would forward the subscription.

When a subscription reaches a node within the region of interest, further forwarding decisions are made based only on the ability of neighbor nodes to reach all two-hop neighbors. In other words, the cone-or-region filter is bypassed and DRP attempts to flood the region to reach as many publishers as possible.

Even after DRP has found a route from the subscriber to the region of interest, it does not prune the tree to eliminate dead-end branches because they afford flexibility in reconnecting routes if the network connectivity were to change. For example, consider node 6 in the previous example. This node is not a forwarder of the subscription because it lies outside the cone and region of interest as described above. However, it receives the subscription at the same time as nodes 7 and 8, and maintains a record of the subscription in its cache of node 3's state. This would become important if node 10 were to move out of contact with nodes 7, 8, and 11, while remaining in the region of interest and maintaining contact with node 6.

In this case, node 6 could serve as the relay node for data from node 10 even though it is not a forwarder of the subscription (because it lies outside the cone where forwarding is allowed). In this case, node 10 would still be able to respond to the subscription. However, node 10 could not *relay* the subscription, for example, to node 13, because it

[2] In dense networks, the buffer angle can be set to 0 degrees.

never receives the subscription as part of its own node state. Node 10 can only sense the subscription in node 6's state and respond if it has a matching publication itself.

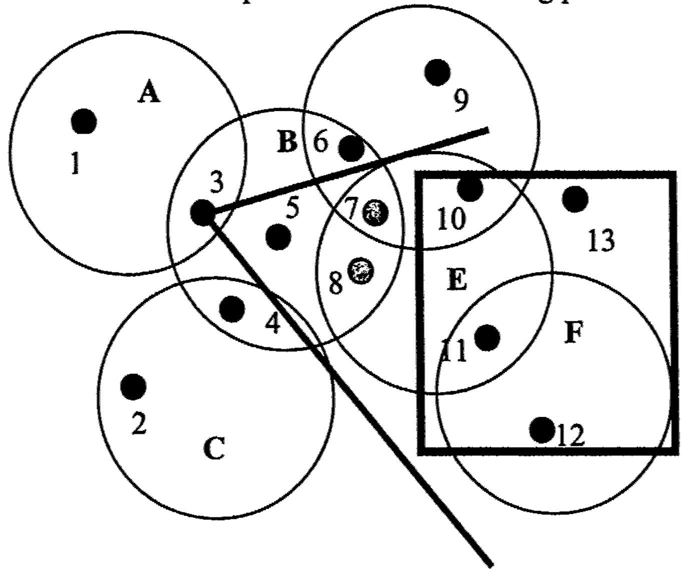

Figure 2-5: The list of nodes that may forward a subscription is limited to those that lie in the specified region of interest (box), or in the cone formed by the forwarding node and the edges of the region of interest. The cone shown here includes a buffer angle that encompasses nodes off the direct path to the target area, increasing the number which may forward subscriptions.

This behavior is the result of illustrating DRP operation in a network where nodes are deployed too sparsely to ensure sufficiently redundant routing trees. In such situations, DRP routing parameters can be tuned to reflect the priorities of the particular deployment and the associated node densities and terrain obstructions. For example, the cone shown in Figure 2-5 can be widened or skipped altogether, increasing reliability and robustness at the expense of higher energy use.

2.3. Subscriptions for Subscriptions

In some circumstances, a node will declare a "subscription for subscriptions." This request has local scope: unlike a data subscription, it is not propagated on the network. This subscription monitors all incoming subscriptions and informs the application if an incoming subscription matches the desired criteria. This capability allows tasking to take place through the subscription mechanism. For example, consider Figure 2-6, in which node 1 subscribes for subscriptions and node 2 subscribes for all acoustic data in the vicinity of node 1, requiring updates every 5 seconds. Node 1 will inform the application of the subscription and tasking request and the application will execute the command (depending on priorities).

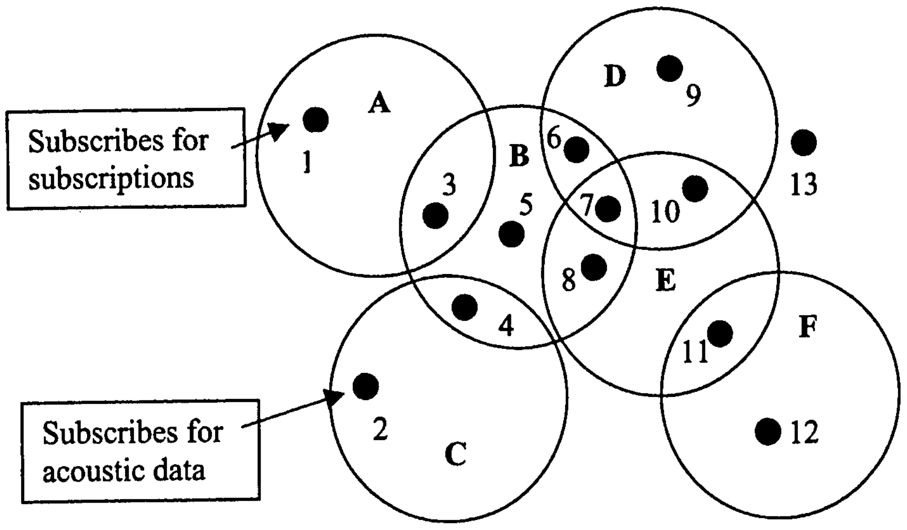

Figure 2-6: Node 1 subscribes for subscriptions, and is thus informed of node 2's interest in acoustic data. Node 2's interest is communicated to node 1 at two levels: the network level, at which node 1 will transmit acoustic data if it has an appropriate publication, and also at the application level, where higher-level functions may be performed, for example tasking a power-inefficient acoustic collection asset.

2.4. Publications

When a node has data to transmit, it publishes a description of what information can be provided. The publication has local scope, so it is not propagated throughout the network. Instead, the publications are locally compared with incoming subscriptions to determine if there is a match. If the sending of data occurs frequently, it is advantageous for the node to set up the publication in advance and have each send reference that publication, avoiding a matching computation for each send (unless the publication or subscription changes).

2.5. Sending/Forwarding Data

After the publications are declared, data can be transmitted. If, in the publisher's neighborhood, there exist one or more subscriptions that match the publication, data is forwarded from the publisher to subscribers along routing trees established immediately after the subscription declaration. Because the routes already exist, there is no computation required to send data, and thus no latency due to routing. If no subscriptions in the neighborhood match the publication, then DRP will conserve resources and not transmit the data.

As an example of data forwarding, consider the network in Figure 2-7. If node 9 acquires data matching the subscription from node 1, it declares a publication as described above and sends a data packet that is forwarded from neighborhood D to neighborhood B and then neighborhood A, along a branch of the routing tree (repeated from Figure 2-4). Once the data arrives in neighborhood A, it is delivered to the subscriber, node 1.

If the routing tree is disrupted (for example by a node drop-out or relocation) the message will not be relayed. Furthermore, DRP does not support the storage of messages, so even if a new path is established during a periodic consistency check, the lost messages will not be re-transmitted unless they are requested (and the publisher is able to comply).

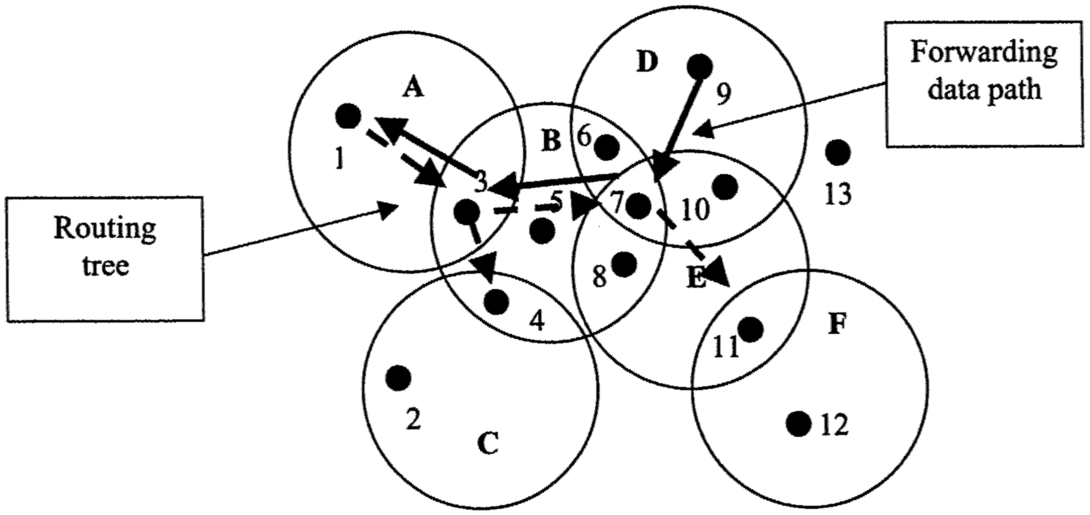

Figure 2-7: Forwarding Data Path

2.6. Node Mobility

DRP routes are robust with respect to node mobility because of the network's distributed node-state database. Each node maintains a remote cache of information about all of its neighbors. This means that when a node moves out of one neighborhood and into another, the effect is localized to the neighboring nodes that must either remove or add the mobile node from their remote node state cache. If the departing node was part of a subscription tree, the remaining nodes have enough information to create a new path. The same is true if the node is destroyed or goes off-line to conserve energy.

As an example, again consider Figure 2-7. If node 7 moves out of neighborhoods B, D, and E into neighborhood F, the remaining neighbors would sense the move and calculate how to reconnect the subscription route, in this case by routing messages by way of nodes 6 and 8. This approach is scalable because the impact of node mobility is limited to the adjacent neighborhoods (just nodes 3, 6, 8, 11 in the case of a move by node 7). While node mobility is supported, *frequent* relocation of nodes to new neighborhoods would cause route maintenance inefficiencies, as routes would have to be recalculated regularly.

If a node's relocation or destruction causes a portion of a routing tree to become completely disconnected from the rest of the network, the subscription is deleted from node states one at a time beginning with the node on the publishing end closest to the break in the routing tree. Each node deletes the subscription from its remote node state records, and updates its node state sequence number. The new node state (without the subscription) is then propagated to all neighbors, including those which forward the subscription. These neighbors update their own node states to delete the subscription, all the way to the end of the tree. On the subscribing end of the tree, the subscription persists in case the network changes again to allow the route to be reestablished.

The requirement of managing these two processes (reconnecting vs. rolling back disconnected routes) is managed by a timer. This timer is set to force a delay before a node can delete subscription information. This delay should be made long enough for the neighborhood on the subscribing end to recognize the break and calculate new forwarders if they exist. The exact delay required to optimize route maintenance is an area where further research is needed.

2.7. Unsubscribe

When a subscribing node has no further need for data in a previously issued subscription, it declares DISINTEREST in that data, deletes the subscription from its node state record, and updates its node state sequence number. The subscription is deleted all along the route in the manner described for handling node mobility.

2.8 Unpublish

When a publishing sensor node is unable to continue to provide data it has published, it issues an unpublish command. This has effect only on the local node; i.e. it is not propagated on the network. Instead, the node simply discontinues sending data, and must establish a new publication if the data becomes available again. The effect on unpublishing node's neighbors is much the same as when the publishing node is destroyed (the only difference being that the node will remain in the neighborhood). Note that the subscriptions persist in the geographic area specified by the subscriber. Consequently, if another publisher becomes able to service the subscription, it can send data immediately.

3.0 CONCLUSION

This report has described the motivation, theory, and algorithms defining the declarative routing protocol (DRP). DRP has application to the area of wireless sensor networking, because of its scalability, bandwidth and energy efficiency, and its capability for ad hoc self-configuration. Present capability and future research directions are described.

APPENDIX A

DECLARATIVE ROUTING PROTOCOL IMPLEMENTATIONS

DRP has been implemented in several versions during its development. While sharing a common code base, the various implementations differ subtly based on the environment in which they run. As an example, multi-hop communication is supported in all versions of DRP, however the communication mechanism servicing the WINS and UDP nodes does not support it. The following sections provide installation, compilation and run-time instructions for use of the different implementations.

	GloMoSim	WindowsNT -UDP	WINS nodes	Linux-UDP
Multi-hop communication	Yes	No	No	Limited
Communication within neighborhood	Yes	Yes	Yes	Yes
Check for uni-directional data links	No	No	No	No
Error Control Coding	No	No	No	No
Error Checking	No	No	Limited	No
Consistency Protocol	Yes	Yes	Yes	Yes
Routing around Dead nodes	Limited	Limited	Limited	Limited
Subscriptions for subscriptions	Yes	Yes	Yes	Yes
Run-time scenario configuration	No	No	No	No

Figure 3-1: Implementation status of DRP features.

Prior to building any version of DRP, you should copy the entire contents of the CD into a directory. This directory is referred to as [CD-copy directory] in the following instructions.

1 Linux-UDP

The Linux-UDP implementation of DRP was created as a desktop testbed for the algorithms involved prior to implementation on the Windows environment native to the Sensoria nodes involved in the SensIT Twenty-Nine Palms experiment [1]. The program executes one of four scenarios specified at run time from a Linux command line interface. To establish communication between two computers running the application, both must reside on the same Ethernet subnet (two invocations on the same computer results in a socket error). The application has subscription/publication information hardcoded into each of the four test cases provided, but reads geographical location information for each node from a file in the directory where the application is invoked. A limitation of this implementation is that every node lies within a single neighborhood and multiple hop communications are never required (without modifying the code to force this to happen).

1.1 Compilation Instructions – drpsocket version

```
COMPILATION:
------------
The source can be compiled fairly easily by executing the following
commands (assuming csh or derivative for the shell, and a valid path to
gmake, the GNU makefile program).

prompt> cd [CD-copy directory]
prompt> source setupenv
prompt> cd drp
prompt> gmake
prompt> cd ..
prompt> gmake

This will build the drp shared library (libdrp.so) and then build the
main program.  You need to "source setupenv" before running so that it
will know where to find libdrp.so.

ORGANIZATION:
-------------
All of the DRP files are located in the drp directory.  Note that most
of those files are common to the Windows CE version, and ifdef's are
used throughout the code to support both unix and windows.

The only files in the top level directory are those used for the test
program.  These files are also used in the UserPlatform code on the
Windows side.

RUNNING:
--------
To run the program:
prompt> ./drpsocket <appNodeID>

The test program will cycle  through and subscribe and unsubscribe to
for both data and incoming subscriptions. Note that the location of the
node is set in the initdata.txt file(which should be edited
appropriately on each node).  The details of the subscriptions are
hardcoded in the test program (since it is just a test program).  Here
are the basics of the subscriptions:

AppNodeID #1:
  Publication:  DEVICE_KEY=*  (EQ_ANY)
  Subscription: subscription for incoming data:
                DEVICE_KEY="video" AND
                LAT <= myLat+10.0 AND
                LON >= myLon+10.0
```

```
AppNodeID #2:
  Publication:  DEVICE_KEY="video"
  Subscription: subscription for incoming subscriptions:
                DEVICE_KEY="*"    (EQ_ANY)

AppNodeID #3:
  Publication:  DEVICE_KEY="video"
  Subscription: None.


AppNodeID #4:
  Publication:  None.
  Subscription#1: subscription for incoming data:
                  DEVICE_KEY="video" AND
                  LAT >= myLat-6.0 AND
                  LON <= myLon+6.0
  Subscription#2: subscription for incoming data:
                  DEVICE_KEY="acoustic" AND
                  LAT >= myLat-6.0 AND
                  LON <= myLon+6.0
```

As the program runs, output to the screen is generated to allow the user to verify that it is working correctly. The output can be turned on/off via a #define in the drpout.cpp file (in the drp directory).

2 GloMoSim

The purpose of this implementation was to compare the performance of DRP with that of other networking approaches such as Bellman-Ford routing within an IP framework, and a "global flood" approach in which all nodes in a distributed network retransmit all messages so as to be sure to reach all potential data providers with every query [14].

For purposes of this comparison, DRP was implemented in the GloMoSim simulation environment (version 1.1.1) running under Redhat Linux version 6.0. Some elements of GloMoSim were modified to better support proper simulation of DRP. Hence, DRP is not yet available under the newest versions of GloMoSim. The code is compiled by a UCLA-developed parallel simulation tool called PARSEC to simulate execution of discrete events such as signaling within DRP.

2.1 PARSEC Information

Prior to compiling the DRP simulation, you must install PARSEC on your machine. The following instructions are for the installation of PARSEC on a RedHat 6.x machine. If you are running a different Unix operating system, you should substitute "redhat-6.0" with the appropriate subdirectory of /usr/local/parsec:

```
prompt> su
prompt> [enter root password]
prompt> cp -r [CD-copy directory]/parsec /usr/local/parsec
prompt> ln -s /usr/local/bin/redhat-6.0/runtime /usr/local/parsec/runtime
prompt> exit
```

2.2 Simulation Compilation Instructions

How to compile and run the glomosim software...

COMPILATION:

The source can be compiled fairly easily by executing the following commands (assuming csh or derivative for the shell).

> IF YOU ARE USING AN OPERATING SYSTEM OTHER THAN REDHAT LINUX 6.x, you must first edit the Makefile in the sensit/main directory so that the PARSEC_ROOT variable corresponds to the operating system you are using. All of the available options are already in the Makefile. All you have to do is comment-out the current definition and uncomment the correct one.

```
prompt> cd [CD-copy directory]/sensit
prompt> source setupenv
prompt> cd main
prompt> gmake
```

This will build the entire glomosim environment and place and executable called "Sim" in the bin directory. You need to "source setupenv" before running so that it will know where to find the parsec libraries (which should be installed first).

ORGANIZATION:

All of the DRP is located in the "transport/drp" directory. Most of these files are shared with the WinCE source tree as well as the drpsocket source tree.

All of the application software that we created is located in the "application" and "application/sensit" directories.

The software is always compiled from the "main" directory - which also contains the basic glomosim architecture. The executable is placed in the "bin" directory and it is called "Sim".

The "bin" directory also contains configuration/input files. There are several of these files like:

```
CONFIG.IN - the default test program (ftp and telnet applications)

DRPTESTCONFIG.IN - a simple test program to have random node
                   placements with one subscriber and one publisher.
ADS.IN - the 10000 node experiment.

SENSIT.IN - the South Lab securing a building scenario.

SMALL.IN - a very simple scenario for testing purposes only.

RRER.IN - the "Random, Random, Everything Random" scenario with
          random placement, subscribers, publishers, frequency of
          sends, etc.
```

```
RUNNING:
--------
To run the program:

prompt> cd [CD-copy directory]/sensit/bin
prompt> ./Sim <configuration/input file>

This program will go through the scenario (as long as indicated in
the configuration file).  It will create the following output files:

GLOMO.STAT - the standard output file that describes the output for
             each layer for each node.  This can get quite large.
GLOMO.ENERGYSTAT - the energy usage over time.  This shows the total
                   system energy usage every .25 seconds.

GLOMO.LATENCYSTAT - the latency for each message.
```

APPENDIX B

Application Programmer's Interface

The version of the application programmer's interface (API) described here is the one that was defined at the time DRP was developed [15]. The SensIT program has since evolved this API to include additional features not described or available in this version.

1 Introduction

SCADDS data diffusion (at USC/ISI) and DRP (at MIT Lincoln Laboratory) are both based on the core concept of subject-based routing. Although there are some fundamental differences between these approaches, we believe that both can be accommodated with the same Network Routing API. This document presents a combined API that both network routing approaches can use.

2 Interest Description

There are a couple of approaches to the details of the interest specification (some of which have been presented to the community). This API document focuses on one approach (after some email discussion). Given that the ***best*** answer to this problem may not be achievable in the near term, we should focus on the getting a solution that will work for the demo in August. Following is an overview of the approach that we will focus on for the August demo.

2.1 Attributes

Data requests and responses are composed of data attributes that describe the data. Each piece of the subscription (an Attribute) is described via a key-value-operator triplet, implemented with `class Attribute`.

• `key` indicates the semantics of the attribute (latitude, frequency, etc.). Keys are simply constants (integers) that are either defined in the network routing header or in the application header.

Allocation of new key numbers will be done with an external procedure to be determined. The initial simple procedure is that MIT and ISI will allocate ranges of the space to projects. Keys in the range 3001-3999 can be allocated temporarily for experimental purposes.

• `type` indicates the primitive type that the `key` will be. This key will indicate what algorithms to run to match subscriptions. For example, checking to see if an INT32_TYPE is EQ is a different operation than checking to see if a STRING_TYPE is EQ. The special case here is the BLOB_TYPE type – this is used for application specific data that can not be used for matching.

• `op` (the *operator*) describes how the attribute will match when two attributes are compared.

The IS operator indicates that this attribute specifies a literal value (the LATITUDE_KEY IS 30.456). Other operators (GE, LE, NE, etc.) mean that this value must match against an IS attribute. Matching rules are below.

Not all operators are appropriate for all keys. All are supported for integers, floats, and strings. Only IS, EQ_ANY, EQ, and NE are supported for blobs.

Keys and operators can be extracted from an attribute via getKey() and getOp() methods.

In addition, attributes have values. Values have some type and contents. Some values also have a length (if it's not implicit from the type). These are the basic constructors:

```
Int32Attribute(int key, int op, int val);
```

```
Float32Attribute(int key, int op, float val);
StringAttribute(int key, int op, char *s);
BlobAttribute(int key, int op, int len, void *s);
```
String is a null-terminated C string, blobs are uninterpreted binary data.

• the method getVal() returns the value of the attribute, appropriately typecast. A void pointer to this data can also be accessed with getGenericVal(). Strings and Blobs also return the length with getLen(). Length of strings does not includes the null byte, but a null byte is guaranteed to be there.

Here's example code creating a set of attributes:

```
NR::Attribute * attrs[5];
attrs[0] = new NR::Int32Attribute(NR::Attribute::CLASS_KEY,
                                  NR::Attribute::IS,
                                  NR::Attribute::INTEREST_CLASS);
attrs[1] = new NR::Float32Attribute(NR::Attribute::LATITUDE_KEY,
                                    NR::Attribute::GT, 54.78);
attrs[2] = new NR::Float32Attribute(NR::Attribute::LONGITUDE_KEY,
                                    NR::Attribute::LE, 87.32);
attrs[4] = new NR::StringAttribute(NR::Attribute::TARGET_KEY,
                                   NR::Attribute::IS,
                                    "tel");
```

2.2 Matching rules

Data is exchanged when there are matching subscriptions and publications and the publisher sends data. Matches are determined by applying the following rules between the attributes associated with the publish (P) and subscribe (S):

```
For each attribute Pa in P, where the operator Pa.op is something other than IS
        Look for a matching attribute Sa in S where Pa.key == Sa.key and Sa.op == IS
                If none exists, exit (no match)
If all are found, repeat the procedure comparing non-IS operators in S against IS operators in P.
        If neither exits with (no match), then there is a match.
```

For example, a sensor would post publish this set of attributes (LAT IS 30.455, LON IS 104.1, TARGET IS tel,), while a user might look for TELs by subscribing with the attribute (TARGET EQ tel), or it might look for anything in a particular region with (TARGET EQ_ANY, LAT GE 30, LAT LE 31, LON GE 104, LON LE 104.5).

As a special exception, if you subscribe to things with CLASS_KEY EQ INTEREST_CLASS, you will also get callbacks with CLASS_KEY IS DISINTEREST_CLASS as descrbibed in the unbsubscribe().

3 Interfaces

Following is a description of each of the methods that are part of the network routing API class.
1. To initialize the NR class, there is a C++ factory called NR::createNR() that will create the appropriate NR class and return a pointer to it.

The prototype of the function is as follows:

```
static NR * NR::createNR();
```

createNR() is a method rather than a constructor so that it can actually create a specific subclass of class NR (one for MIT and one for ISI-W).

createNR() method will create any threads needed to insure callbacks happen.

2. The application declares interest in via the `NR::subscribe` interface. This function accepts a list of

interest declarations and uses this information to route data to the necessary nodes.

The prototype of the function is as follows:

```
handle NR::subscribe(
int num_interest_attrs,
const Attribute * const interest_declarations[],
const NR::Callback * cb);
```

• `num_interest_attrs` specifies the number of interest attributes are associated with this subscription.

• `interest_declarations` is an array containing the elements that describe the subscription information (see notes above about the class type).

• `cb` indicates the class that contains the implementation of the method to be called when incoming data (or tasking information) matches that subscription. The class inherits from the following abstract base class:

```
class Callback
{
public:
virtual void recv(int num_attrs, const Attribute * const data[],
handle h) = 0;
};
```

After subscriptions are diffused throughout the network, data arrives at the subscribing node. The `recv()` method (implemented by the application) is called when incoming matching data arrives at the network routing level. This method is used to pass the incoming data and tasking information up to the caller. See entry 7 below for more detail about callbacks.

Subscribe returns a handle (which is an identifier of the interest declaration/subscription). This handle can be used for a later unsubscribe() call. If there is an error in the subscription call (based on local information, not the propagation of the interest), then a value of –1 will be returned as the handle.

Note that the condition that no data matches the attributes is not considered an error condition---if the application requires or expects data to be published, application-level procedures must determine conditions that might cause no data to appear. (As one example, if the goal is to contact a few nearby sensors, the app might start with a small region around it and expand or contract that region until the expected number of sensors reply.)

Subscribes are used to get both data and to find out about subscriptions from other nodes. To get data, (if you're a data sink) subscribe with CLASS_KEY IS INTEREST_CLASS. This subscription will propagate through the network and the callback will eventually trigger when matching data returns.

To find out about interests (if you're a sensor or data source), subscribe including CLASS_KEY EQ INTEREST_CLASS. The callback you provide will be called for each new subscription (with CLASS_KEY EQ INTEREST_CLASS) and whenever a subscription goes away due to unsubscribe or detection of node failure (timeout), resulting in callbacks with CLASS_KEY EQ DISINTEREST_CLASS. (This is a special exception to the matching rules.) These callbacks will be made at least once for each unique expression of interest. If multiple clients express interest in exactly the same thing, this callback will occur at least once.

3. The application indicates that it is no longer interested in a subscription via the `NR::unsubscribe` interface. This function accepts a subscription handle (originally returned by subscribe()) and removes the subscription associated with that passed handle.

The prototype of the function is as follows:

```
int NR::unsubscribe(
handle subscription_handle)
```

• `subscription_handle` is a handle associated with the subscription that the application wishes to unsubscribe to. It was returned from the `NR::subscribe` interface.

The return value indicates success/failure. If there is a problem with the handle, then an error code of –1 is returned. Otherwise, 0 is return for normal operation. If an unsubscribe is issued for a subscription that contains a task, then the each node that has received that task will be informed of the unsubscribe.

4. The application also indicates what type of data it has to offer. This is done via the `publish` function. The prototype of the function is as follows:

```
handle NR::publish(
int num_pub_decls,
const Attribute * const publication_decls[])
```

• `num_pub_decls` specifies the number of interest attributes that are associated with this publication.

• `publication_decls` is an array of publication declarations (see notes above about the class type).

This function returns a handle (which is an identifier of the publication) that will be later used for unpublish(). If there is an error in the publication call, then a value of –1 will be returned as the handle. This handle is used later when unpublishing or sending data.

5. The publish interface has a matching unpublish interface (`NR::unpublish`).

The prototype of the function is as follows:

```
int NR::unpublish(
handle publication_handle)
```

• `publication_handle` is the handle associated with the publication that the application wishes to unpublish. It was returned from the `NR::publish` interface.

The return value indicates success/failure. If there is a problem with the handle, then an error code of –1 is returned. Otherwise, 0 is return for normal operation.

6. After the publications are set up the application can send data via the `NR::send` function. This function will accept a single attribute to send associated with a publication handle (the handle used in the associated `NR::publish` function call). This method does not guarantee delivery, but the system will make reasonable efforts to get it to its destination. Probability of message loss will be not much greater than probability of sending the message with the underlying WINSng send calls.

The prototype of the function is as follows:

```
int NR::send(
handle publication_handle,
const Attribute * data_set)
```

• `publication_handle` is the handle that is associated with the block of data that was sent. In other words, the `data_set` matches the publication indicated with the publication handle.

• `data_set` is the block of data associated with a publication (as defined by the `publication_handle`).

The return value indicates success/failure. If there is a problem with the arguments, then an error code of –1 is returned. Otherwise, 0 is return for normal operation. If there is currently no one interested in the message (no matching subscription), then it will not consume network resources.

Future versions (post August 2000) of the API may have a sendAttrs() call that accepts multiple attributes.

7. After the subscribe is issued from the application thread, the application thread can wait for the reception of information via the `NR::Callback::recv()` method.

The network routing system will provide a thread to make callbacks happen. Since there is only one such thread in the system, the callback function should return reasonably quickly. If the callback needs to do some compute-bound or IO-bound task, it should signal another thread.

The receiveOne() method previously in the API has been removed (since the network routing system now provides the callback thread). This behavior is may to change when we shift to a multi-process architecture.

4 C++ Header

Following is the above descriptions in the Network Routing API class and the associated support classes. A single instance of the NR class will need to be created per node.

1. The publish interface has a matching unpublish interface (`NR::unpublish`).

The prototype of the function is as follows:

```
int NR::unpublish(
handle publication_handle)
```

• `publication_handle` is the handle associated with the publication that the application wishes to unpublish. It was returned from the `NR::publish` interface.

The return value indicates success/failure. If there is a problem with the handle, then an error code of –1 is returned. Otherwise, 0 is return for normal operation.

2. After the publications are set up the application can send data via the `NR::send` function. This function will accept a single attribute to send associated with a publication handle (the handle used in the associated `NR::publish` function call). This method does not guarantee delivery, but the system will make reasonable efforts to get it to its destination. Probability of message loss will be not much greater than probability of sending the message with the underlying WINSng send calls.

The prototype of the function is as follows:

```
int NR::send(
handle publication_handle,
const Attribute * data_set)
```

• `publication_handle` is the handle that is associated with the block of data that was sent. In other words, the `data_set` matches the publication indicated with the publication handle.

• `data_set` is the block of data associated with a publication (as defined by the `publication_handle`).

The return value indicates success/failure. If there is a problem with the arguments, then an error code of –1 is returned. Otherwise, 0 is return for normal operation. If there is currently no one interested in the message (no matching subscription), then it will not consume network resources.

Future versions (post August 2000) of the API may have a sendAttrs() call that accepts multiple attributes.

3. After the subscribe is issued from the application thread, the application thread can wait for the reception of information via the `NR::Callback::recv()` method.

The network routing system will provide a thread to make callbacks happen. Since there is only one such thread in the system, the callback function should return reasonably quickly. If the callback needs to do some compute-bound or IO-bound task, it should signal another thread.

The receiveOne() method previously in the API has been removed (since the network routing system now provides the callback thread). This behavior is may to change when we shift to a multi-process architecture.

5 C++ Header

Following is the above descriptions in the Network Routing API class and the associated support classes. A single instance of the NR class will need to be created per node.

```
// Match Operator values
enum operators { IS, LE, GE, LT, GT, EQ, NE, EQ_ANY };
// with EQ_ANY, the val is ignored
Attribute();
Attribute(int key, int type, int op, int len, void *val = NULL);
Attribute(const Attribute &rhs);
~Attribute();
int32_t getKey() { return key_; }
int8_t getType() { return type_; }
int8_t getOp() { return op_; }
int16_t getLen() { return len_; }
void * getGenericVal() { return val_; }
protected:
int32_t key_;
int8_t type_;
int8_t op_;
int16_t len_;
void *val_;
};
// these interfaces allow type-safe attribute initialization
class Int32Attribute : public Attribute {
public:
Int32Attribute(int key, int op, int val);
int getVal() { return *(int *)val_; }
};
class Float32Attribute : public Attribute {
public:
Float32Attribute(int key, int op, float val);
float getVal() { return *(float *)val_; }
};
class StringAttribute : public Attribute {
public:
StringAttribute(int key, int op, char *s);
char * getVal() { return (char *)val_; };
};
class BlobAttribute : public Attribute {
public:
BlobAttribute(int key, int op, int len, void *s);
void * getVal() { return val_; };
};
class Callback
```

```
{
public:
virtual void recv(int num_attrs,
const Attribute * const data[],
handle h) = 0;
};
// Factory to create an NR class specialized for ISI-W or MIT-LL's
// implementation (whichever is compiled in).
static NR * createNR();
virtual handle subscribe(int num_interest_attrs,
const Attribute * const interest_declarations[],
const NR::Callback * cb) = 0;
virtual int unsubscribe(handle subscription_handle) = 0;
virtual handle publish(int num_publication_attrs,
const Attribute * const publication_declarations[])
= 0;
virtual int unpublish(handle publication_handle) = 0;
virtual int send(handle publication_handle,
const Attribute * data_set) = 0;
};
#endif // _NR_H
```

6 API Walk Through

This walkthrough considers, at a high-level, what happens in the SensIT August 2000 scenario.
The db front end running on a node (say, node A) wants information about TELs and expresses this interest by calling NR::subscribe with the following attributes:

CLASS_KEY IS INTEREST_CLASS
SCOPE_KEY IS GLOBAL_SCOPE
LONGITUDE_KEY GE 10
LONGITUDE_KEY LE 50
LATITUDE_KEY GE 20
LATITUDE_KEY LE 40
TASK_FREQUENCY_KEY IS 500
device_type EQ seismic
TARGET_KEY IS TEL
TARGET_RANGE_KEY LE 50
CONFIDENCE_KEY EQ_ANY *
TASK_NAME_KEY IS detect_track
TASK_QUERY_DETAIL_KEY IS [query_byte_code]

Note the distinction between IS which specifies a known value and EQ, which specifies a required match. All of the comparisons are currently ANDed together. The query parameters that interact with network routing (for example, lat/lon) must be expressed as attributes, but some other parameters may appear only in application-specific fields (such as the query_byte_code in this example).

NR::subscribe returns right away with a handle for the interest. Because this interest has a global scope, network routing forwards it to the neighboring nodes, that proceed using the pre-defined rules.
Nodes with sensor(s) tell network routing about the type of data they have available by publishing. An application on a node (say, node B) would call NR::publish with the following attributes:

CLASS_KEY IS DATA_CLASS
SCOPE_KEY IS NODE_LOCAL_SCOPE
LONGITUDE_KEY IS 10
LATITUDE_KEY IS 20
TASK_NAME_KEY IS detectTrack
TARGET_RANGE_KEY IS 40
device_type IS seismic

After receiving the handle, the application can start sending data with the NR::send command. For example, each detection it might invoke send() with the attribute (CONFIDENCE_KEY IS .8) and then handle from the publish command. This attribute (CONFIDENCE_KEY) would be associated with the other attributes associated with publish handle and eventually delivered to anyone with a matching subscribe. Initially, since the node has no matching interest, the data will not propagate to other nodes. When interest arrived data would begin to propagate.

In some cases, the application may wait to start sensing until it has been *task*ed, either to avoid doing unnecessary work, or to use the parameters in the interest to influence what it looks for. In this case, the sensor would get the task by subscribing to interests with NR::subscribe and the following attributes:

CLASS_KEY EQ INTEREST_CLASS
SCOPE_KEY IS NODE_LOCAL_SCOPE
LONGITUDE_KEY IS 10
LATITUDE_KEY IS 20
TASK_NAME_KEY IS detectTrack
TARGET_RANGE_KEY IS 40
device_type IS seismic

(The only difference in these attributes with the publish call is in the CLASS key and operator.) The callback associated with this subscribe will then be called each time a new subscription arrives or goes away. Arrivals will have CLASS_KEY IS INTEREST_CLASS, unsubscribes will have CLASS_KEY IS DISINTEREST_CLASS.)

7 API Usage Examples

Step 1: Initialization of Network Routing

```
{
// This code is in the initialization routine of the
// network routing client.
.
.
.
// Create an NR instance. This will create (if necessary) the
// NR instance and return a pointer to that instance. This
// will only be created once per node. Save this somewhere
// where it can be used from wherever data needs to be sent.
netrouting = NR::createNR();
// Setup the publications and subscriptions for this NR client.
// Note the details of the following calls are found in step 3
setupPublicationsOnSensorNode(netrouting);
setupSubscriptionsOnSensorNode(netrouting);
.
.
.
// Do any other initialization stuff here...
}
```

Step 2: Create callbacks for incoming data/subscriptions

```
// In a header file somewhere in the client setup two callback
// classes. One to handle incoming data and the other to handle
// incoming subscriptions/tasking.
class DataReceive : public NR::Callback
{
public:
void recv(int num_attrs,
const NR::Attribute * const *data,
handle h);
};
class TaskingReceive : public NR::Callback
{
public:
void recv(int16 num_attrs,
const NR::Attribute * const *data,
handle h);
};
// In the appropriate C++ file, define the following methods
void DataReceive::recv(
int num_attrs,
const NR::Attribute * const *data,
handle h)
{
// called every time there is new data.
// handle it.
}
// In the appropriate C++ file, the define the following methods
void TaskingReceive::recv(
int num_attrs,
const NR::Attribute * const *data,
handle h)
{
// Handle incoming tasking.
}
```

Step 3: Setup publications and subscriptions

```
void setupPublicationsOnSensorNode(
NR * netrouting)
{
// Setup a publication with 2 attributes. One attribute indicates
// that this publication is of type data (not
// interest/subscription) the second attribute indicates that
// this node has an acoustic device and can provide that type
// of data.
NR::Attribute *attrs[5];
attrs[0] = new Int32Attribute(NR::Attribute::CLASS_KEY,
NR::Attribute::IS, NR:Attribute::DATA_CLASS);
attrs[1] = new StringAttribute(DEVICE_KEY, // user defined
NR::Attribute::IS,
"seismic");
attrs[2] = new Float32Attribute(LATITUDE_KEY, // user defined
NR::Attribute::IS,
45.0);
attrs[4] = new Float32Attribute(LONGITUDE_KEY, // user defined
NR::Attribute::IS,
103.21);
// publish these attributes save the pub_handle somewhere like in
// the private data.
pub_handle = netrouting->publish(4, attrs);
if (pub_handle == -1)
{
// ERROR;
}
}
// create two subscriptions (one for the sensor node and the other
// for the user node.
void setupSubscriptionsOnSensorNode(
NR * netrouting)
{
// (1) the first subscription indicates that I am interested in
// receiving tasking subscriptions for this node.
//
// Each of the subscriptions will have its own callback to separate
// incoming subscriptions and incoming data.
// ****************
// SUBSCRIPTION #1: First setup the subscription for
// tasking interests...
NR::Attribute attrs[2];
attrs[0] = new Int32Attribute(CLASS_KEY,
NR::Attribute::EQ,
NR::Attribute::CLASS_INTEREST);
attrs[1] = new StringAttribute(DEVICE_KEY, // user defined
NR::Attribute::EQ_ANY,
NULL);
// Create the callback class that will be used to
// handle subscriptions that match this subscription.
TaskingReceive * tr = new TaskingReceive();
// subscribe and save the handle somewhere like in
// the private data.
task_sub_handle = netrouting->subscribe(2, attrs, tr);
if (task_sub_handle == -1)
{
// ERROR;
}
|}
void setupSubscriptionsOnUserNode(
NR * netrouting)
{
// (2) the second subscription indicates that I am interested in
// nodes that have seismic sensors in a particular region
// and have
// them run a task call detectTel (with some specific query
// byte code attached). This task instructs the nodes to send
// data back.
// ****************
// SUBSCRIPTION #2: First setup the subscription for
// tasking subscriptions...
NR::Attribute attrs[9];
attrs[0] = new Int32Attribute(CLASS_KEY,
NR::Attribute::IS,
NR::Attribute::CLASS_INTEREST);
attrs[1] = new StringAttribute(DEVICE_KEY, // user defined
NR::Attribute::IS,
"seismic");
attrs[2] = new Float32Attribute(NR::Attribute::LATITUDE_KEY,
NR::Attribute::GE,
44.0);
attrs[4] = new Float32Attribute(NR::Attribute::LATITUDE_KEY,
NR::Attribute::LE,
46.0);
attrs[5] = new Float32Attribute(NR::Attribute::LONGITUDE_KEY,
NR::Attribute::GE,
103.0);
attrs[6] = new Float32Attribute(NR::Attribute::LONGITUDE_KEY,
NR::Attribute::LE,
104.0);
attrs[7] = new StringAttribute(NR::Attribute::TASK_NAME,
NR::Attribute::IS,
"detectTel");
attrs[8] = new StringAttribute(NR::Attribute::TASK_QUERY_DETAIL_KEY,
NR::Attribute::IS,
query_byte_code.len,
&query_byte_code.contents);
// Create the callback class that will be used to handle
// subscriptions that match this subscription.
DataReceive * dr = new DataReceive();
// subscribe and save the handle somewhere like in
// the private data.
```

```
data_sub_handle = netrouting->subscribe(6, attrs, dr);
if (data_sub_handle == -1)
{
// ERROR;
}
}
```

Step 4: Sending data

```
{
.
.
// When one of the clients have data to send, then it will use
// the NR::send method. Assume that there is acoustic data to
// send (matching the publish call above). The data is found
// in the variable adata with the size of adata_size.
Attribute * attr = new BlobAttribute(DATABLOCK_KEY, // user defined
Adata_size,
&adata);
if (netrouting->send(pub_handle, attr) == -1)
{
// ERROR;
}
delete attr;
.
.
}
```

Step 5: Receiving data

(the RecieveData callback is called every time data arrives)

APPENDIX C

Overview of DRP Source Code

This appendix contains comments about the functionality implemented within each file of the DRP source code. It is intended to serve as a guide to software at a lower level than that provided in the main body of this report.

What's in DRP files:

drp.h: Declarations for packet headers used in DRP's node-to-node communications.

drpCfront: A transition layer between C and C++ code, used for GloMoSim.

drpbase: These files declare and define the DRPBase class. In the code, the DRPBase structure creates the DRPLocalNode structure and indirectly the other structures that form the DRP communications interface. DRPLocalNode is derived from DRPBase via DRPNode.

Note that only in GloMoSim can there be more than one DRPBase structure per computer. (Each such structure represents one virtual node.) In other cases there can be only one DRPBase class, and attempts to call DRPBase::createNode() will return the existing DRPBase class.

The code in the DRPBase class primarily dispatches incoming consistency protocol requests to the code (primarily in DRPLocalNode) which deals with them.

drpcomm*:

drpcommethernet:
Creates threads that maintain consistency between nodes for the Ethernet (IP) data transfer interface.

drpcommglomosim:
Manages inter-node communications within GloMoSim.

drpcomminterface:
Dispatches communications between DRP and other nodes. It also creates the threads for the consistency protocol (in initCommInterface), and creates the DRPBase class for the given node number.

drpcp: Implements the DRP consistency protocol; basically, it is responsible for implementing the distributed node state and keeping the node state consistent. This module is one of the fundamental modules that controls DRP's operation.

If DEBUG_DRPCP is defined, many debugging outputs are enabled in drpcp.

DRPCP::receiveNodeStateHeartBeat processes heartbeats from other nodes. These heartbeats indicate the current version number of the state of the node - if the version number indicated in the heart beat is not the same as the version number of the state in the local node's record, the local node will send a NAK to request a new copy of the node's state. The heartbeat will also reschedule the timeout for that node (when a node times out, it is deleted from the local node's state). If the received node state version number is less than the node state version number recorded by the local node, the node is presumed to have restarted. To erase and recreate the node's record, the heartbeat is ignored until the node times out.

DRPCP::receiveNodeStateNAK is called if a NAK arrives. If a NAK arrives, the new node state is not sent out immediately - this allows several NAKs from neighborhood nodes to be processed simultaneously.

DRPCP::execute dispatches all of the periodic operations of the consistency protocol. Whenever a timer runs out, the execute method is called.

drpdatamarshall:
Defines serialization operators for various data types used by DRP.

drpfilter: Contains two main pieces of code. First, it contains code to match subscriptions against publications. FilterInfo::attrsMatch compares two attributes. FilterInfo::subFiltersPass takes an the subscription represented by the instance of the FilterInfo class and compares it against a provided publication publication (passed as a parameter) on the local host.

The second piece of code in drpfilter is to determine if a given node should forward a subscription. A node will *only* forward a subscription if it is within the 2D cone formed by the previous-hop node and the outer borders region of interest. The main function call for this is FilterInfo::withinRegionOrCone, which is given the location of the previous-hop node and the local node. This function is passed the previous-hop location and the local node's location; the subscription region used is the one represented by the particular instance of the FilterInfo class.

drplocalnode:

This file contains code that relates to the creation and maintenance of subscriptions and routes, and therefore deserves close scrutiny.

The heart of this code is in DRPLocalNode::receiveIncomingData, which receives an incoming DRP packet and deals with it, forwarding or passing up to the application as is required.

The procedure for choosing to forward the data is as follows:

1. Find the subscription record that matches the incoming data.
2. Check if it's a local subscription. If so, pass up to application.
3. Check for forwarding nodes.
4. Forward to the forwarding node, or broadcast if multiple forwarding nodes exist.

setSubscription and unsetSubscription are used respectively to add and delete subscriptions. setLocalSubscription sets up a subscription that is not forwarded across the network. Note that you can subscribe to data locally; this will forward any data that arrives at the node to the user, without explicitly requesting that the data be sent on the network. (It will appear only if the node is on the forwarding path for another, similar subscription – a "piggy-back" of sorts.)

Bugs: In unsetSubscription, the line "_nodeState.subs.erase(it)" is commented out. This is a memory leak and means that deleted subscriptions won't be properly deallocated - this causes problems in the long term. Unfortunately due to iterator quirks in C++ this is difficult to fix (and has not been fixed). A (probably) related problem arises when a reused subscription is encountered. In this case, the routing fails to find a next-hop node.

drpnode: A virtual class used as a superclass for drplocalnode and drpremotenode. Also includes part of the constructor for the DRPLocalNode class, which creates the consistency protocol associated with the local node. Contains several utility functions to read out subscriptions, publications, and state from the node record.

Generally, this class contains everything that a remote node can do - add and remove subscriptions, maintain state, and pass information on new and erased subscriptions up to applications running on the local node. There is little functionality specific to remote nodes, so most of what is implemented by remote nodes is implemented here.

drpnre: The NRE class contains the Ethernet / IP interface code. The broadcast address is hardcoded near the end of drpnre.cpp. This introduces an endian dependency in the code which can be corrected using htonl() to do the conversion.

The code reads a set of dotted-decimal IP addresses from the file "neighbors" if it exists. This set of addresses is compared against all incoming (broadcast) DRP packets, and any packet from an address not on this list is dropped. This can be used to implement the concept of "neighborhoods" on an Ethernet where all devices are visible to all others. If the file doesn't exist, all addresses are visible. The DISP_RECV compile time flag, if defined, prints a record of all packets as they arrive, and an indication of whether they were dropped.

A known problem in this routine is that when determining the node ID number, the first Ethernet device (eth0) is used, even though the actual wireless interface is eth1. Therefore, it is important to make sure that the Ethernet addresses are unique. Remember to change the address in the system configuration of the node.

drpout: A debugging class used to print messages in Windows. Some recent changes are not supported here; a lot of the debugging messages probably won't show up if the code is compiled under Windows.

drpremotenode:
DRPRemoteNode holds the state of remote nodes. It's mostly used as a data structure; DRPRemoteNode doesn't do much on its own (most of its functionality is inherited from the DRPNode class). Also it doesn't implement all the virtual functions defined by DRPNode. (The only code implemented is searching for and updating remote subscriptions.)

drproute: The DRPRoute class (declared in libdrproute.h) is split up into several .cpp files:

drproutedecision.cpp:
drproutedecision.cpp contains the code that implements the procedure for rating, sorting, and selecting boundary nodes to cover all of the two-hop nodes. This file can be changed to implement energy, connectivity, and link quality biases.

drprouteprint.cpp:
This file contains code to display the results of the route decision process. (call DRPRoute::printResults to do this.)

drproutetwohop.cpp:

This file contains code that creates and maintains a list of all nodes that can be reached in two hops from the local node, and the nodes through which they can be reached. This is used by drproutedecision to determine if all two-hop nodes are covered by a proposed set of forwarding nodes. The list of forwarders is automatically recalculated if the list of two-hop nodes or their forwarders changes.

drprouteupdate.cpp:

This file contains the core of the DRP routing code. The main role of the code in this file is to determine when new routes need to be generated – with the intention of spending time doing route calculations only when something has changed, because the routing is relatively expensive. DRPRoute::updateSelf is the main function of this file; it accepts a list of new and removed subscriptions, and updated list of remote nodes, and then updates all of the records to account for the changes.

interface: This module contains many implementation-specific functions, including setting up mutexes and timers and retrieving random numbers. For some reason, getRandomNumber always returns 0.5. This only affects delays in drpcp - therefore it's more likely that multiple NAK's will be sent instead of just one, but should otherwise have no adverse effects.

nrdrp: This module contains the application software interface to DRP. It also includes the code that generates handles. Handles have 24 bits of the node number and 8 bits for a sequence number. This means there can only be 256 subscriptions for a node at a time.

The function NR::generateHandle has been changed to reuse subscription numbers only when it runs out of numbers. This improves (but does not fix) unrelated problems in deleting and recreating subscriptions caused by the incomplete deletion process.

timers: A class to implement timer callbacks. The main function here is "TimersManager::runTimer," which is called from interface.c "TimerCheck" periodically (via a thread).

REFERENCES

[1] Srikanta Kumar and David Shepherd, "SensIT: Sensor Integration Technology for the Warfighter," in *Proceedings of the Fourth Annual Conference on Information Fusion*, August 7-10, 2001.

[2] Chalermek Intanagonwiwat, Ramesh Govindan and Deborah Estrin, "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks", To appear in *Proceedings of ACM Mobicom '00*, August 2000.

[3] John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin and Deepak Ganesan, "Building Efficient Wireless Sensor Networks with Low-Level Naming," to appear, *Proceedings of the Symposium on Operating System Principles,* Lake Louise, Banff, Canada, October 2001.

[4] "Energy Efficient Technologies for the Dismounted Soldier", NRC Study, 1997.

[5] Xiang Zeng, Rajive Bagrodia, Mario Gerla, "GloMoSim: A Library for Parallel Simulation of Large-scale Wireless Networks," Proceedings of the 12th Workshop on Parallel and Distributed Simulation – PADS '98," May 26-29, 1998.

[6] Rajive Bagrodia, Richard Meyer, Mineo Takai, Yu-an Chen, Xiang Zeng, Jay Martin, Ha Yoon Song, "Parsec: A Parallel Simulation Environment for Complex Systems," *IEEE Computer*, October 1998.

[7] T.-H. Lin, H. Sanchez, H.O. Marcy, and W.J. Kaiser, "Wireless Integrated Network Sensors (WINS) for Tactical Information Systems," in *Proceedings of the 1998 Government Microcircuit Applications Conference.*

[8] S. Singh, M. Woo, and C.S. Raghavendra. "Power-Aware Routing in Mobile Ad Hoc Networks". In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, pp 181-190, October, 1998.

[9] C. Perkins and P. Bhagwat. "Routing over Multihop Wireless Network of Mobile Computers". *SIGCOMM '94: Computer Communications Review*. pp 234-24, Oct 1994.

[10] C. Perkins, E. Royer. "Ad-Hoc On Demand Distance Vector (AODV) Routing". Charles E. Perkins and Elizabeth M. Royer., *Proc. of the Second IEEE Workshop on Mobile Computing Systems and Applications* (WMCSA) New Orleans, LA, February 1999, pp 90-100.

[11] D. Johnson and D Maltz. "Dynamic Source Routing in Ad-Hoc Wireless Networks". In *Computer Comunications Review – Proceedings of SIGCOMM '96*, Aug 1996.

[12] V. Park and S. Corson. "Temporally-Ordered Routing Algorithm (TORA) version 1 functional specification." *draft-ietf- manet-tora-spec-02.txt* (work in progress).

[13] J. Navas and Tomasz Imielinski, "GeoCast - Geographic Addressing and Routing", *Proc. of the Third ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'97)*, Budapest, Hungary, Sept 1997.

[14] D. Coffin, D. Van Hook, S. McGarry and S. Kolek, "Declarative Ad Hoc Sensor Networking," SPIE Vol. 4126 Integrated Command Environments, 31 July 2000, pp. 109-120.

[15] D. Coffin, D. Van Hook, J. Heidemann, and F. Silva "Network Routing Programmer's Interface and Walk Through, Draft 7.6, August 2000.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (*Leave blank*)	2. REPORT DATE 28 February 2003	3. REPORT TYPE AND DATES COVERED Project Report

4. TITLE AND SUBTITLE Declarative Routing Protocol Documentation	5. FUNDING NUMBERS C — F19628-00-C-0002
6. AUTHOR(S) P. Boettcher, D. Coffin, R. Czerwinski, K. Kurian, M. Nischan, D. Sachs, G. Shaw, and D. Van Hook	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Lincoln Laboratory, MIT 244 Wood Street Lexington, MA 02420-9108	8. PERFORMING ORGANIZATION REPORT NUMBER ECCS-1
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Dr. Sri Kumar DARPA 3701 N. Fairfax Drive Arlington, VA 22203-1714	10. SPONSORING/MONITORING AGENCY REPORT NUMBER ESC-TR-2001-086

11. SUPPLEMENTARY NOTES

None

12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (*Maximum 200 words*)

This report documents the motivation, present capability, and theoretical promise of the Declarative Routing Protocol (DRP) developed at MIT Lincoln Laboratory as part of the DARPA Sensor Information Technology (SensIT) program. DRP was developed as a means of enabling distributed wireless sensors to configure themselves into a scalable ad hoc network and respond in an energy-efficient way to asynchronous requests for sensor information. Conventional networking approaches are generally not adequate for such applications because of energy constraints, reliability and scalability requirements and the greater variability in topology present compared with traditional fully-wired or last-hop wireless (remote to base station) networks. DRP operates within these constraints by exploiting query-supplied data descriptions to control network routing and resource allocation.

14. SUBJECT TERMS routing dynamic networks publish/subscribe	15. NUMBER OF PAGES 50
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Same as Report	19. SECURITY CLASSIFICATION OF ABSTRACT Same as Report	20. LIMITATION OF ABSTRACT Same as Report